MAZE: Towards Automated Heap Feng Shui

Yan Wang[1,2], *Chao Zhang*[3,4] ✉, *Zixuan Zhao*[1,5], *Bolun Zhang*[1,5], *Xiaorui Gong*[1,5], *Wei Zou*[1,5]

[1] *{CAS-KLONAT*[*], *BKLONSPT*[†]*}, Institute of Information Engineering, Chinese Academy of Sciences* [2]*WeiRan Lab, Huawei Technologies*
[3]*BNRist & Institute for Network Science and Cyberspace, Tsinghua University*
[4]*Tsinghua University-QI-ANXIN Group JCNS*
[5]*School of Cyber Security, University of Chinese Academy of Sciences*
wangy0129@gmail.com, chaoz@tsinghua.edu.cn, beraphin@gmail.com, {zhangbolun, gongxiaorui, zouwei}@iie.ac.cn

Abstract

A large number of memory corruption vulnerabilities, e.g., heap overflow and use after free (UAF), could only be exploited in specific heap layouts via techniques like heap feng shui. To pave the way for automated exploit generation (AEG), automated heap layout manipulation is demanded.

In this paper, we present a novel solution MAZE to manipulate proof-of-concept (POC) samples' heap layouts. It first identifies heap layout primitives (i.e., input fragments or code snippets) available for users to manipulate the heap. Then, it applies a novel `Dig & Fill` algorithm, which models the problem as a Linear Diophantine Equation and solves it deterministically, to infer a primitive operation sequence that is able to generate target heap layout.

We implemented a prototype of MAZE based on the analysis engine S2E, and evaluated it on the PHP, Python and Perl interpreters and a set of CTF (capture the flag) programs, as well as a large micro-benchmark. Results showed that MAZE could generate expected heap layouts for over 90% of them.

1 Introduction

Automated exploit generation (AEG) is playing an important role in software security. Software vendors could utilize it to quickly evaluate the severity of security vulnerabilities and allocate appropriate resources to fix critical ones. Defenders could learn from synthetic exploits to generate IDS (Intrusion Detection System) rules and block potential attacks.

Existing AEG solutions [1, 2, 3, 4, 5] are effective at exploiting stack-based or format string vulnerabilities, which are rare in modern systems [6]. Few could handle heap-based vulnerabilities, which are more common. Heap-based vulnerabilities in general can only be exploited in specific heap layouts. For instance, a common way to exploit a heap overflow is placing another object with sensitive code pointers (e.g. VTable or function pointer) after the overflow object. However, heap objects' lifetime and heap layouts are dynamic and hard to determine or manipulate. In practice, it requires abundant human efforts and techniques, e.g., heap feng shui [7].

To manipulate heap layouts, in general we have to find primitives that are able to interact with heap allocators first, and then assemble them in a specific way by (de)allocating objects of specific sizes in a specific order.

1.1 Recognize Heap Layout Primitives

A heap layout operation primitive is a building block for heap layout manipulation, which can be (re)used by users to interact with target programs' underlying heap allocators. Programs usually do not expose such primitives directly.

SHRIKE [8] and Gollum [9] focus on generating exploits for language interpreters (e.g., Python and PHP), and mark statements in input scripts (a group of input bytes) as heap layout manipulation primitives. SLAKE [10] generates exploits for Linux kernels, and marks system calls as heap layout manipulation primitives. These primitives trigger heap (de)allocation operations and can be assembled freely. *But they are not applicable to most other applications.* For instance, file processing applications neither accept freely assembled input files nor provide APIs for users to invoke.

Furthermore, to precisely manipulate heap layouts, we also need to understand (1) the semantics of primitives, e.g., the count and size of (de)allocations performed in each primitive; and (2) the dependencies between primitives, e.g., the order between them. Failing to do so would cause further primitives assembly process ineffective, as shown in SHRIKE [8].

Our solution: Note that, most applications are driven by a certain form of events, including messages, user interactions, data fragments, and network connections etc.. Code snippets dispatched in the event processing loops usually are reentrant and could be utilized to manipulate heap layouts. We therefore extend primitives to such reentrant code snippets, and use static analysis to recognize them, and analyze their semantics and dependencies accordingly.

1.2 Assemble Heap Layout Primitives

To generate expected layouts, we further need to assemble the set of recognized heap layout primitives in a specific way.

[†]Key Laboratory of Network Assessment Technology, CAS
[‡]Beijing Key Laboratory of Network Security and Protection Technology

SHRIKE [8] applies a random search algorithm, which is inefficient and undecidable, to find primitive sequences that could place two specific objects next to each other. Gollum [9] further improves the efficiency with an evolutionary algorithm. SLAKE [10] utilizes the characteristics of kernel heap allocators, and proposes a customized algorithm to place victim objects after vulnerable objects.

However, they fail to address several challenges. First, instead of relative offsets between two objects, the expected heap layout could be too complicated to model. For instance, to perform an *unsafe unlink* attack [11], two chunks are needed to allocate before and after the overflowed chunk, and therefore offsets between three objects are required. Second, each heap operation primitive may allocate and deallocate multiple objects at the same time, and even interfere with other primitives. Therefore, primitives may have side effects (i.e., noises), and make it challenging to assemble. For instance, the success rate of SHRIKE [8] drops dramatically when the number of noises grows. Lastly, different heap allocators have different heap management algorithms, causing different heap layouts even with the same sequence of heap operation primitives. Therefore, allocator-specific solutions (e.g., SLAKE) cannot be simply extended to other applications.

Our solution: We reduce the heap layout manipulation problem to a basic problem of placing a specific object `O` at a specific position `P`, and propose a `Dig & Fill` algorithm. At the time of allocating `O`, if the location `P` is occupied by other objects, then we will `dig` proper holes in advance to accommodate them. Otherwise, if `P` is empty but `O` still falls into other holes, then we will `fill` those holes in advance.

Each heap layout operation primitive may yield a number of `dig` and `fill` operations. Thus, we setup a `Linear Diophantine Equation` [12], to calculate the count of each primitive required. By solving this equation deterministically, we infer the heap interaction primitive sequence that could generate the target layout.

1.3 Results

In this paper, we presented an automated heap layout manipulation solution MAZE to address the aforementioned challenges. We built a prototype based on the binary analysis engine S2E [13], and evaluated it in three different settings: (1) 23 vulnerable CTF (Capture The Flag) programs, (2) the PHP interpreter with 5 known vulnerabilities, targeting 10 different heap layouts respectively, as well as the Python and Perl interpreter with 10 vulnerabilities, and (3) 3000 randomly generated test cases with large primitive noises.

Results showed that, MAZE has a high success rate and efficiency. It successfully converted 16 CTF programs' heap layouts into exploitable states, efficiently generated expected heap layouts for the PHP, Python and Perl in all cases, and generated expected heap layouts for the random test cases with a success rate of over 90%.

In summary, we have made the following contributions:

- We proposed a novel automated heap layout manipulation solution MAZE, able to generate complicated heap layouts (e.g., with multi-object constraints) for a wide range of heap allocators, facilitating automated exploit generation.
- We proposed a new and general type of heap layout operation primitives, and proposed a solution to recognize and analyze such primitives.
- We proposed a novel `Dig & Fill` algorithm to assemble primitives to generate expected heap layouts, by solving a `Linear Diophantine Equation` deterministically.
- We pointed out primitive noise is not the primary bottleneck of automated heap feng shui, and made MAZE robust against primitive noises.
- We implemented a prototype of MAZE [1], and demonstrated its effectiveness in CTF programs, language interpreters, and synthetic benchmarks.

2 Background

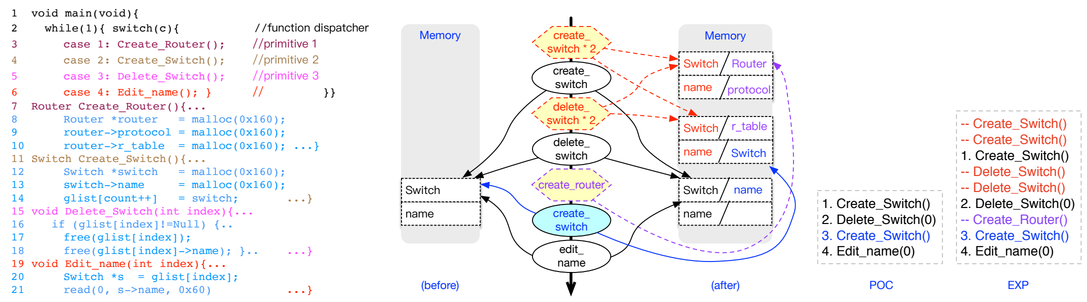
2.1 Automated Exploit Generation (AEG)

AEG for Stack-based Vulnerabilities: Early AEG solutions rely on deterministic recipes, e.g., the classical methods to exploit stack-based or format string vulnerabilities, to automatically generate exploits. Heelan et al.[1] proposed to utilize dynamic taint analysis to generate control-flow-hijack exploits when given crashing POC inputs. Avgerinos et al. coined the term AEG [2] and developed an end-to-end system to discover vulnerabilities and exploit them with symbolic execution. They further extended the solution to support binary programs in Mayhem [3]. Similarly, starting from the crashing point, CRAX [5] symbolically executes the program to find exploitable states and generate exploits.

AEG for Heap-based Vulnerabilities: Unlike stack-based vulnerabilities, heap-based vulnerabilities in general are harder to exploit. Repel et al. [14] utilizes symbolic execution to find exploit primitives that are derived from heap chunk metadata corruption, and then tried to generate exploits using a SMT solver. Revery [15] utilizes a layout-oriented fuzzing and a control-flow stitching solution to explore exploitable states when given a non-crashing POC. HeapHopper [16] utilizes symbolic execution to discover exploitation techniques for heap allocators in a driver program. PrimGen [17] utilizes symbolic execution to find a path from the crashing point to a potentially useful exploit primitive. Most of these solutions can not manipulate heap layouts, and only work when the given POC sample's heap layout is good to go.

AEG for Various Targets: FUZE [18] utilizes fuzzing to find different dereference sites of dangling pointers in system calls, and facilitates the process of kernel UAF exploitation. Kepler [19] facilitates kernel exploit generation by automatically generating a "single-shot" exploitation chain. The solution teEther [20] extends AEG to vulnerabilities in smart

[1] We open source MAZE at https://github.com/Dirac5ea/Maze to facilitate the research in this area.



```
1  void main(void){
2    while(1){ switch(c){              //function dispatcher
3       case 1: Create_Router();      //primitive 1
4       case 2: Create_Switch();      //primitive 2
5       case 3: Delete_Switch();      //primitive 3
6       case 4: Edit_name(); }        //              }}
7  Router Create_Router(){...
8       Router *router    = malloc(0x160);
9       router->protocol = malloc(0x160);
10      router->r_table  = malloc(0x160); ...}
11 Switch Create_Switch(){...
12      Switch *switch    = malloc(0x160);
13      switch->name      = malloc(0x160);
14      glist[count++]    = switch;          ...}
15 void Delete_Switch(int index){...
16    if (glist[index]!=Null) {..
17      free(glist[index]);
18      free(glist[index]->name); }..     ...}
19 void Edit_name(int index){...
20      Switch *s   = glist[index];
21      read(0, s->name, 0x60)            ...}
```

POC

1. Create_Switch()
2. Delete_Switch(0)
3. Create_Switch()
4. Edit_name(0)

EXP

-- Create_Switch()
-- Create_Switch()
1. Create_Switch()
-- Delete_Switch()
-- Delete_Switch()
2. Delete_Switch(0)
-- Create_Router()
3. Create_Switch()
4. Edit_name(0)

Figure 1: Example vulnerability and the heap layout manipulation solution. Hexagons with dashed edges are primitives to insert.

contracts. FLOWSTITCH [21] aims to generate data-oriented exploits, which could reach information disclosure without breaking the control flow. Ispoglou et al. [22] proposed the BOP, which could utilize basic blocks as gadgets along valid execution paths to generate data-oriented exploits.

2.2 Automated Heap Layout Manipulation

Heap layout manipulation is a critical challenge of AEG, recognized as the *heap likelihood inference issue* in [23, 24]. Several solutions have been proposed in recent two years.

SHRIKE [8] randomly assembles program input fragments (script statements) to search for inputs that could generate expected layouts. Gollum [9] applies an evolutionary algorithm to improve efficiency. However, many applications' input fragments cannot be freely assembled together to yield valid inputs. Besides, different primitives (input fragments) could have dependencies and side effects (noises), greatly lowering the success rate and efficiency of SHRIKE and Gollum.

SLAKE [10] is another solution able to manipulate heap layouts. It targets only Linux kernel vulnerabilities, and applies an algorithm specific to the simple and deterministic Linux slab allocator to assemble system calls. However, it has a narrow application scope. Most applications neither have direct interaction interfaces like system calls, nor have simple heap allocators. It also suffers from the primitive noise issue.

2.3 Problem Scope

2.3.1 Applicable Programs

Our solution MAZE is only applicable to event loop driven programs. Most programs are driven by user input events or messages, and usually have function dispatchers enclosed in loops to handle these events. For example, network interaction programs are driven by commands in connections, language interpreters are driven by sentences in scripts.

2.3.2 Applicable Heap Allocators

Our solution MAZE can be applied to multiple allocators as long as they obey four rules as below:

Rule 1: Deterministic Behavior. A same sequence of heap operations will yield a same heap layout, if a same initial heap layout is provided. The majority of allocators are deterministic, such as ptmalloc[25] and dlmalloc[26]. Some allocators are deterministic under some conditions, such as jemalloc[27] in single thread environment.

Note that, the allocator can have non-deterministic initial state. MAZE will utilize heap spraying [28] to fill all holes in the initial state and reach a deterministic state. After that, new chunks could be (de)allocated as if they are in an empty heap.

Rule 2: Freed memory areas first. Allocators reuse one of the recently freed areas to serve the allocation request, rather than finding a new area from inventory. This strategy can improve memory utilization efficiency and is adopted by most allocators, such as ptmalloc, dlmalloc and jemalloc.

Rule 3: Freed areas of same size first. Allocators prefer to choosing the freed areas of same size to serve the allocation request. This strategy is usually used to reduce memory fragments and widely adopted.

Rule 4: Re-allocation order is deterministic. Freed memory chunks are usually kept in linked lists, and re-allocated to new objects in certain order. Some allocators use the lastly freed chunk to serve the new allocation request, i.e., following the LIFO (Last-In-First-Out) policy, e.g. fastbin in ptmalloc[25], while some others follow the FIFO (First-In-First-Out) policy, e.g. normal chunk in ptmalloc[25].

3 Motivation Example

We will introduce the overview of our heap layout manipulation solution MAZE, with a running example shown in Figure 1. There is a UAF vulnerability, where the `Switch` object is freed at line 17 but its pointer is kept in the global list and referenced again at line 21.

3.1 Expected Memory Layout Generation

First, we need to analyze the vulnerability in POC automatically, there are many sanitizers [15, 29, 30, 31, 32] proposed for this purpose. As mentioned in Revery[15], dynamic analysis can be used to identify the vulnerability point and exceptional object. In this example, we can know the vulnerability is a UAF and the `Switch` object is the exceptional object.

Then, the expected memory layout can be generated according to the properties of the vulnerability. In this example, a controllable object (e.g. `switch->name`) should take the freed exceptional object's position, to hijack the reference of the freed object `s` at line 21, and drive it to write to an

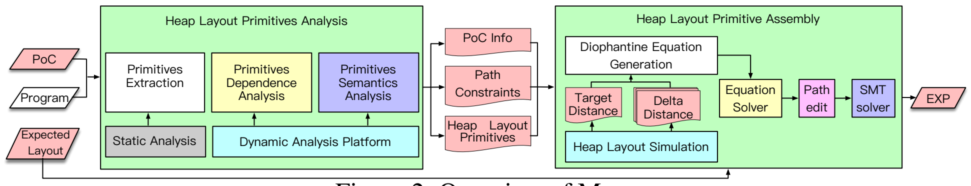


Figure 2: Overview of Maze

address `s->name` controlled by the attacker, yielding arbitrary memory writes. For other types of vulnerabilities, e.g. the buffer overflow, an exploitable object should be placed next to the overflowed object. Using existing solutions, the expected exploitable memory layout can be generated. And it will be an input for MAZE.

3.2 Memory Layout Manipulation

This part is the main focus of MAZE. Given the expected layout (i.e., placing a `switch->name` at the freed `switch` object's position), simply invoking `Create_Switch` may yield another uncontrollable `switch` object at the target location, unable to control the memory write pointer (i.e. POC in Figure 1). Instead, the adversary could manipulate the heap layout in another way to place a controllable `switch->name` at the target position. MAZE aims at finding such a manipulation scheme automatically. Figure 2 shows the overview of MAZE, which consists of two major components discussed as follows.

3.2.1 Heap Layout Primitives Analysis

In this part, taking the program and POC as inputs, MAZE will extract primitives in them. Heap layout primitives (e.g., `Create_Switch`) are the building blocks for heap layout manipulation. Different from existing solutions, we extend heap layout primitives to reentrant code snippets.

Primitives Extraction: Reentrant code snippets usually exist in function dispatchers that are enclosed in loops. Therefore, we could utilize the code structure characteristic to recognize candidate heap layout primitives, via static analysis.

Primitives Dependency Analysis: Some reentrant code snippets may depend on other snippets. For instance, a snippet responsible for freeing an object has to wait for another snippet to create the object first. By analyzing the pre-condition and post-condition of each code snippet, we could recognize such dependencies and merge them into one primitive.

Primitives Semantics Analysis: To better assemble primitives, we have to understand the semantics of each primitive, especially the size of objects (de)allocated in each primitive using taint analysis and symbolic execution.

Example: In this example, given the program, by analyzing its code structure, we could recognize several primitives at line 3, 4 and 5. Further, we could infer that primitive `Delete_Switch` depends on the primitive `Create_Switch`, and therefore group them as a new primitive.

Given the POC, MAZE also extracts the heap primitives used in POC's execution trace (i.e. POC info in Figure 2), to infer the original memory layout and the inserting points.

3.2.2 Heap Layout Primitives Assembly

The inputs of this part are heap primitives, POC info, path constraints and expected layout. MAZE will utilize heap primitives to manipulate POC's layout (infered from the POC info) to the expected layout and generate an exploit using a constraint solver.

Intuition: The problem of generating an expected heap layout could be modelled as placing a group of objects in a group of memory addresses. Without loss of generality, we will first consider placing one object `O` into one target address `P`. As shown in Figure 3, there are two cases to handle.

Dig case: As shown in Figure 3(a), at the time of allocating the target object `O`, the target address `P` could be taken by noise objects (e.g., `O'`). In this case, we will dig (multiple) holes before allocating noise objects `O'`, by adding primitives that could free objects of proper sizes, to accommodate noise objects `O'` and leave the hole `P` to the target object `O`.

Fill case: As shown in Figure 3(b), at the time of allocating the target object `O`, the target address `P` could be empty, but `O` still falls into other holes. In this case, we will fill (multiple) holes before allocating `O`, by adding primitives that could allocate objects, and leave the hole `P` to the target object `O`.

Following this `Dig & Fill` guidance, we could add proper heap layout primitives into the program execution trace to yield expected layouts.

Standard fill (or dig) operation If a fill (or dig) operation contains only one allocation (or deallocation), and the size equals to the size of `O` (or `P`), we call it a standard fill (or dig) operation. Obviously, a standard fill (or dig) operation can fill (or dig) only one hole with the same size of `O` (or `P`).

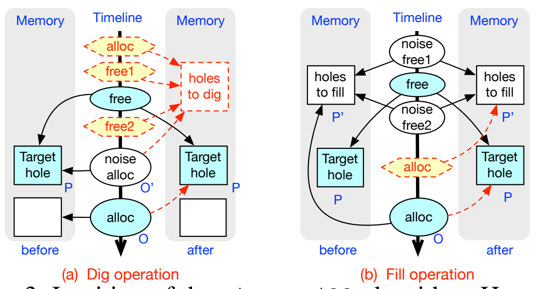


Figure 3: Intuition of the `Dig & Fill` algorithm. Hexagons are heap layout primitives to invoke. Only one of `free1` and `free2` exists (before or after the creator of the target hole), depending on the heap allocator's strategy (i.e., FIFO or LIFO).

Target Distance Analysis As long as we add enough standard fill (or dig) operations into the program execution trace, the target object `O` can be placed into the target address `P`.

To analyse how many standard fill (or dig) operations are needed, we craft a shadow program with the same heap allocator, which only handles (de)allocations and takes commands from the analyzer. According to the heap primitives in POC's execution trace (i.e. the POC info), the analyzer will instruct the shadow program to free or allocate objects of specific sizes. Therefore it will derive the original memory layout and determine whether dig or fill operation is required. Then the analyzer will keep inserting standard fill or dig operations until the target object `O` is placed into the target address `P`. If `d` standard dig operations are required, the `Target Distance` is set to `+d`. Otherwise, if `d` standard fill operations are required, the `Target Distance` is set to `-d`.

In this example, one standard dig operation is needed (i.e., one hole should be dug) so that `Create_Switch` can place `switch->name` at the target position. In other words, the `Target Distance` of original layout of POC is +1.

Delta Distance Analysis Heap layout primitives are usually not standard. And we evaluate how many standard dig or fill operations a primitive is equivalent to. We also utilize the shadow program to evaluate the `Target Distance`. Assuming the `Target Distance` before and after inserting a primitive are `d1` and `d2` respectively, then the `Delta Distance` of this primitive is `d2-d1`.

In this example, the `Delta Distance` of primitives `Create_Switch`, `Create_Router` and `Delete_Switch` (combining with its dependant Create_Switch) are +2, +3 and -2 respectively.

Linear Diophantine Equation Generation Given the `Target Distance` to reduce and the `Delta Distance` of each primitive, we could set up a `Linear Diophantine Equation` [12] to calculate the count of each primitive required to reduce the `Target Distance` to zero (i.e., satisfying the expected layout constraint).

In this example, assuming the count of these primitives are x_1, x_2, x_3, we could build a `Linear Diophantine Equation` as follows:

$$\begin{cases} 2x_1 + 3x_2 - 2x_3 + 1 = 0 \\ x_1, x_2, x_3 \geq 0 \end{cases}$$

By querying solvers like Z3[33], we could get a solution: $(x_1 = 0, x_2 = 1, x_3 = 2)$. Therefore, we will add one `Create_Router` and two `Delete_Switch` primitives to the program execution trace. Lastly, we could get the heap layout operation primitive sequence as shown in Figure 1.

Exploit Generation By inserting the inferred primitive sequence to the original program trace, MAZE can generate a program trace that could yield an expected heap layout. As a result, we can utilize techniques like symbolic execution and constraint solving to generate exploit samples. And this is the final output of MAZE.

3.3 Full Chain Exploit Composition

Given the exploitable memory layout generated by MAZE, several other challenges need to be addressed in order to generate a full chain exploit. For instance, defenses like ASLR do not hinder MAZE from manipulating heap layout but could block it from generating working exploits. So we need to find a solution to bypass such deployed security mechanisms. These challenges are out of the scope of MAZE.

4 Heap Layout Primitives Analysis

Heap layout primitives are the building blocks for heap layout manipulation. However, applications usually do not expose interfaces for users to directly interact with the underlying heap allocators. Existing solutions utilize repeatable input fragments and reentrant system calls as primitives to manipulate the heap layout, having limited application scope.

Note that, most applications are driven by different forms of events (e.g., messages, commands, connections), and usually have loops to dispatch event handlers (code snippets). These handlers are reentrant and could be utilized as primitives to interact with underlying heap allocators. We therefore extend heap layout primitives to such reentrant code snippets.

4.1 Primitives Extraction

Since primitives are reentrant code snippets in function dispatchers enclosed in loops, we could analyze the code structure (i.e., control flow graph) to recognize primitives.

In practice, the loop body of a function dispatcher is a switch statement or a group of nested if-else branch statements with related conditions. Each one of such switch cases or branches usually represents a reentrant event handler.

Following the algorithm [34], we could first identify candidate loops in target applications. Then, we could recover potential switch statements and nested if-else statements in candidate loops, following [35, 36]. Lastly, we mark switch cases or if-else branches that have memory (de)allocation operations as candidate reentrant code snippets.

If the count of candidate snippets in a loop exceeds a threshold, then this loop is a candidate function dispatcher and the reentrant code snippets are marked as candidate primitives. The threshold should be more than one, in order to distinguish from simple loops, e.g., a loop for memory write or reads. In our experiment, we take a heuristic value 5 as the threshold. If the program is complicated and have many candidate loops, we can increase the threshold to reduce candidate primitives.

Primitives Extraction for Interpreters. MAZE also supports extracting primitives for language interpreters, e.g., PHP and Python. Similar to previous solutions, MAZE utilizes a fuzzer to generate test cases, and extracts each sentence in scripts as a potential heap layout primitive.

4.2 Primitive Semantics Analysis

The semantics of each primitive, e.g., the size of heap (de)allocation, is critical for precise heap layout manipulation. Therefore, we need to analyze primitive semantics first.

Path Symbolization: Note that, a primitive is a code snippet, which could also have internal branches and form may paths. It is infeasible to analyze all paths to compose the semantics of a primitive. Fortunately, we only care about heap (de)allocations, and many basic blocks do not have such operations. We therefore propose a novel technique *path symbolization* to merge paths with similar heap operations.

First, we remove exception handling paths, since they cannot serve as heap layout operation primitives. Second, for two basic blocks in a primitive, if all sub-paths between them have no heap operations, we will merge all these sub-paths together as a *symbolic sub-path* and mark their basic blocks as symbolic. Therefore, all paths of a primitive could be drastically reduced to several symbolized paths, each consists of a sequence of non-symbolic blocks and symbolic sub-paths.

Symbolic Execution: For each symbolized path, we will evaluate its semantics with symbolic execution. We first utilize symbolic execution to find a path from the program starting point to the entry of this path, then iterate basic blocks in this path as follows. When a symbolic block is iterated, we will perform path traversal for the corresponding symbolic sub-path, and aggregate the symbolic execution results. When a non-symbolic blocks is iterated, symbolic execution is performed as normal. An aggregated symbolic execution result will be yielded for each symbolized path.

Note that, loops will cause path explosion issue for symbolic execution as well. We mitigate this issue by unfolding loops up to a limited number, e.g., a heuristic value 4.

Heap Allocation Size Inferrence After performing symbolic execution, we could get the primitive's allocation size. If the size is symbolic, then we utilize the Satisfiability Modulo Theories (SMT) solver Z3 [33] to derive its value range.

Note that, a primitive with variable heap allocation sizes can be used as a set of different primitives. For instance, if we could allocate objects of size 0x20, 0x40 and 0x60 in a primitive `P` with different inputs, then we could get three different primitives `P_0x20`, `P_0x40`, and `P_0x60`, which share the same code snippet but have different heap effects. They could be used to satisfy different object layout constraints.

4.3 Primitives Dependency Analysis

Primitives may depend on other primitives. For instance, a file read operation has to take place after a file open operation. Therefore, we have to analyze such dependencies and group primitives with dependencies together.

Pairing Allocation and Deallocation: Given an object, it is useful to recognize when it is allocated and freed. We applied a customized taint analysis to pair heap allocations and deallocations. More specifically, we assign a unique birthmark tag to the object at each heap allocation site, and propagate the tags along program execution. At each heap deallocation site, we will examine the object's tag, and link it to the corresponding heap allocation site.

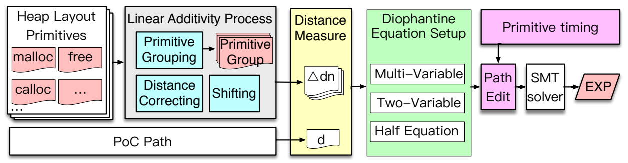


Figure 4: Overview of Heap Layout Primitive Assembly

Recognizing Path Dependency: Some sub-paths of a primitive may depend on another primitive. As shown in Figure 1, `Delete_Switch` relies on the global variable `glist`, which is set by another primitive `Create_Switch`.

Since the primitives exist in a function dispatch, the most common dependency is maintained by variables (including global variables) visible to the function dispatcher. We therefore examine the branch conditions of each primitive, and check if it relies on some variables that are modified by other primitives. If so, the former primitive is likely to depend on the latter. Further, we will execute the former dependent primitive without the latter primitive, and validate whether it will crash. If yes, we can confirm that the dependent primitive relies on the latter primitive.

5 Heap Layout Primitive Assembly

Given the set of recognized heap layout manipulation primitives, the next step is assembling them in a specific way and adding them to the original program path taken by the POC sample, to generate the expected heap layout.

5.1 Overview

Figure 4 shows the workflow of our solution. At the core, a `Dig & Fill` algorithm (§5.2) is applied to manipulate the heap layout. To determine how many dig and fill primitives are needed, we will measure (§5.3) `Target Distance` of the expected layout and `Delta Distance` of each primitive, and setup a `Linear Diophantine Equation` (§5.4) accordingly, then solve it deterministically to resolve the count of each primitive. In some cases, we cannot simply add the distances of two primitives together. Therefore, we will pre-process primitives to guarantee their linear additivity (§5.4.2).

Given the count of each primitive, we will add them to the original POC path in an order guided by the primitive timing principle (§5.5), and yield a new path that is likely to have expected layout. For language interpreters (e.g., PHP), MAZE inserts primitives (e.g., sentences) to the original POC, and adjusts variable names in sentences based on the dependency.

Lastly, we will utilize symbolic execution to generate path constraints of the new path and collect data constraints of primitives (e.g., allocation size), and then query the SMT solver Z3 [33] to generate exploit samples when possible.

For simplicity, we will start from discussing one object layout constraint, and extend it to multi-object layout constraints later (§5.6). Moreover, we will discuss the factors that affect the success rate of heap layout manipulation in §5.7.

5.2 `Dig & Fill` Algorithm

As explained in Section 3.2.2, we will manipulate the heap layout following a `Dig & Fill` algorithm. At a high level, there are three cases:

- **Win:** At the time of allocating the target object `O`, it could be placed exactly in the target hole `P`.
- **Dig:** At the time of allocating `O`, `P` is occupied by other objects. In this case, we will dig some holes, by invoking proper primitives, before allocating the occupying objects. As a result, the occupying objects will fall into the holes we prepare, then leave the target hole `P` to object `O`.
- **Fill:** At the time of allocating `O`, `P` is empty, but `O` still falls into other holes. In this case, we will fill those trap holes, by invoking proper primitives, before allocating `O`, and drive it to take the target hole `P`.

Therefore, placing a target object at a target hole can be simplified as digging or filling multiple memory holes. On the other hand, each heap layout primitive could be modelled as a combination of multiple dig operations and fill operations. So, to generate the expected layout, we just need to figure out the number of each primitive required and their order.

5.3 Distance Measurement

To figure out how many holes have to be dug or filled, we will evaluate the `Target Distance` of the target object to the target hole. On the other hand, we will evaluate how many holes could be dug or filled by each primitive by evaluating its `Delta Distance`.

5.3.1 Heap Layout Simulation and Monitoring

Note that, heap allocators are too complicated to model offline (e.g., via symbolic execution). The sizes of allocations and deallocations in POC execution trace and heap primitives are usually different. And the splitting and merging mechanisms make it almost impossible to derive the distance statically.

Instead, MAZE regards the allocator as a black box and uses a shadow program to simulate the heap layout. The simple shadow program has the same heap allocator as the target application, and only performs heap operations, such as `malloc 0x20` and `free [obj_id]`. The MAZE analyzer will instruct the shadow program to simulate a sequence of heap operations, and then scan the shadow program's heap layout to infer the target application's.

5.3.2 `Target Distance` Measurement

To evaluate the `Target Distance`, the analyzer will instruct the shadow program to perform standard fill (or dig) operations. Specifically, the shadow program will solely allocate (i.e., fill) or allocate then deallocate (i.e., dig) objects of proper sizes (equal to the size of `O` and `P` respectively), i.e., only fill (or dig) holes with the same size of `O` (or `P`).

If `d` standard dig operations are needed, then the `Target Distance` is `+d`. Otherwise, if `d` standard fill operations are needed, the distance is `-d`. In other words, `Target Distance` means how many standard dig or fill operations are needed to create the expected memory layout.

5.3.3 `Delta Distance` Measurement

To measure the `Delta Distance` of a primitive, we will evaluate the `Target Distance` before and after invoking this primitive.

To simplify the evaluation, we will set the `Target Distance` to 0 before invoking primitives, i.e., the target object falls to the target hole in the shadow program. Then we perform the same heap operations as the primitive in the shadow program, and calculate the new `Target Distance`, and denote it as the `Delta Distance` of this primitive. If the `Delta Distance` is `-d` or `+d`, the primitive is therefore called dig or fill primitive.

It should be noted that a primitive is usually not a standard fill (or dig) operation, e.g. it may contain multiple allocations (i.e. noises) or the size of heap operations are not equal to `O` (or `P`). So `Delta Distance` means how many standard dig or fill operations the primitive is equivalent to. For example if the `Delta Distance` is `-n`, it means the primitive can be equivalent to `n` standard dig operations. But the primitve may either dig `n` standard holes, or dig one hole which is big enough to places `n` objects.

5.4 `Linear Diophantine Equation` Setup

Given the `Delta Distance` of each primitive Δd_1, Δd_2, $\Delta d_3 ... \Delta d_n$, we will first calculate the count of each primitive x_1, $x_2, x_3 ... x_n$, in order to reduce the `Target Distance` from `d` to zero. Therefore, we could generate a `Linear Diophantine Equation` as follows.

$$\begin{cases} \Delta d_1 x_1 + \Delta d_2 x_2 + \Delta d_3 x_3 + ... + \Delta d_n x_n + d = 0 \\ x_1, x_2, x_3 ... x_n \geq 0 \end{cases} \tag{1}$$

5.4.1 Existence of Solutions

Note that, if there are no dig or fill primitives, only a small number of heap layouts (i.e., `d`) could be satisfied. On the other hand, this case is rare in practice. Therefore, we will assume there are always at least one dig and one fill primitive. Following the well-known Bezout's Lemma, we could then infer the following theorem. The proof is listed in Appendix A.

Theorem 1. *The aforementioned equation has a non-negative solution (x_1, x_2, ..., x_n), if and only if (1) the greatest common divisor* gcd(Δd_1, Δd_2, ...Δd_n) *divides* `d`*, and (2) there are at least one positive and one negative integer in (Δd_1, Δd_2, ...Δd_n), i.e., there are at least one dig and one fill primitive.*

5.4.2 Linear Additivity of Primitives

Ideally, the `Delta Distance` of each primitives could be linearly accumulated. However, it may be not true in practice, causing the `Linear Diophantine Equation` nonsense.

Instead of analyzing the allocations and deallocations in a primitive, MAZE only calculates how many standard dig or fill operations the primitive is equivalent to. But the sizes of allocations or deallocations in primitives are not always standard (i.e., not equal to the size of `O` and `P`). And due to the splitting mechanism of allocators, the `Delta Distance` may not be linearly accumulated.

After an in-depth analysis, we found three types of heap operation mainly cause the nonlinear additivity. 1) `Bad alloc`: its size is not equal to the target allocation `O`'s. 2) `Bad hole`: its size is not equal to the target hole `P`'s. 3) `Little alloc`: its size is less than half of `P`'s.

For example, if the primitive has a `little alloc`, `P` will be cut into a smaller hole, and `O` can not be placed at `P` again, which means the hole is filled. So the `Delta Distance` is measured as +1. But if the primitive is added again, the `little alloc` will be placed at the rest part of `P`, i.e. `Delta Distance` is 0. Therefore this primitive does not have linear additivity. The detailed analysis can be found in Appendix B. We propose several methods, including `grouping`, `correcting` and `shifting`, to address this problem.

Take the `grouping` technique as an example, it is used to derive primitives which are linearly accumulated with themselves. In general, it puts multiple primitives in a group, which becomes linearly accumulative with itself. If a dig (or fill) primitive is not self linear accumulated, its `Delta Distance` will not be constant. Then MAZE keeps inserting this primitive to a clean memory layout (i.e. `Target Distance` = 0). Because memory holes or allocations increase periodically, the `Delta Distance` of this primitive will also change periodically. Then MAZE puts all the primitives in one cycle together and derives a new primitive, which has linear additivity. Details could be also found in Appendix B.

Then MAZE further ensures different primitives have linear additivity. After `grouping`, MAZE will search for fill primitives that do not contain `bad alloc` or `little alloc`. These fill primitives are linearly accumulated with almost arbitrary dig primitives. Then MAZE can generate a `Multi-variable Diophantine Equation` to derive the expected memory layout. MAZE also searches dig primitives which contain no `bad hole`, and the following process is the same. If no such primitives are available, MAZE will select a pair of fill and dig primitives, and utilize the `grouping` technique again. Then MAZE can generate a `Half Diophantine Equation` without `Target Distance`. If the `Delta Distance` of two primitives are coprime, the `Linear Diophantine Equation` always has solutions. Then MAZE will keep inserting the fill (or dig) primitive to shift the layout state until `O` is placed at `P`. More detail could be found in Appendix C.

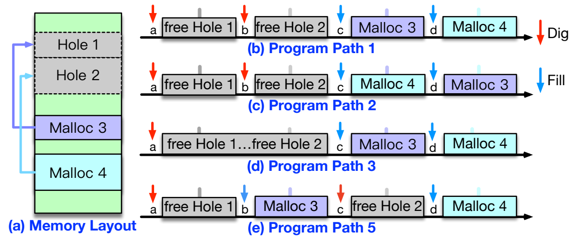

Hole 1
Hole 2
Malloc 3
Malloc 4
(a) Memory Layout
a free Hole 1 b free Hole 2 c Malloc 3 d Malloc 4
Dig
(b) Program Path 1
a free Hole 1 b free Hole 2 c Malloc 4 d Malloc 3
Fill
(c) Program Path 2
a free Hole 1…free Hole 2 c Malloc 3 d Malloc 4
(d) Program Path 3
a free Hole 1 b Malloc 3 c free Hole 2 d Malloc 4
(e) Program Path 5

Figure 5: Examples of two-object position constraint.

5.5 Primitive Timing

Given a solution of the `Linear Diophantine Equation`, we will add those primitives to the original POC path, to generate expected heap layout. However, different primitive placements cause different heap layouts. Therefore, we should determine where to put these primitives in the POC path. Assuming the heap layout constraint is placing a target object `O` at a target hole `P`, we will address this problem as follows.

Adding a fill primitive: In this case, as shown in Figure 3(b), we place the fill primitive (the `alloc` hexagon) before the allocation of the target object (the `alloc` oval), to fill noise holes and drive the target object towards the target hole.

Adding a dig primitive: As shown in Figure 3(a), a dig primitive usually consists of an allocation sub-primitive (the `alloc` hexagon) and a deallocation sub-primitive (the `free1` or `free2` hexagon). To avoid the allocation sub-primitive taking the target hole, we will place it before the target hole creator (the `free` oval). Furthermore, we will add extra allocation primitives around this sub-primitive to isolate the newly created object, to avoid heap chunk merging when this new object is freed later. Then, we will place the deallocation sub-primitive right before the target hole creator if the heap allocator adopts a FIFO policy, or place it after the target hole creator if a LIFO policy is adopted, to free the newly created object (i.e., dig a higher priority hole) to accommodate the noise allocation and leave the target hole to the target object.

5.6 Multi-Object Layout Constraint

A multi-object heap layout constraint could be reduced to placing a group of objects to a group of holes. Ideally, we could decouple the constraints and solve the constraint of each object individually. However, the dig and fill operations for one object could influence another object's placement, making it infeasible to apply the divide and conquer algorithm. Existing solutions all fail to address this challenge.

5.6.1 Motivation Example

Figure 5(a) shows an example two-object layout constraint, where the allocations `Malloc 3` and `Malloc 4` should take the positions `Hole 1` and `Hole 2` respectively. Assume the heap allocator adopts a FIFO policy here. Figure 5(b)(c)(d)(e) are four different POC program paths to manipulate. Here, we will discuss the path in Figure 5(b) first.

According to the aforementioned primitive timing policy, dig primitives will be added before the creator of target hole (if the allocator adopts the FIFO policy). However, dig primitives instrumented before the creator of `Hole 1` will change the `Target Distance` of the object created by `Malloc 4` to its target `Hole 2`, but not the vice versa.

Similarly, fill primitives will be added before the allocation of target object. However, fill primitives instrumented before `Malloc 3` will change the `Target Distance` of the object created by `Malloc 4` to its target hole, but not the vice versa.

Then, assume there are only one dig and one fill primitive in the target application, with `Delta Distance` Δd_{dig} and Δd_{fill} respectively. Assume the `Target Distance` of hole 1 and 2 are d_1 and d_2 respectively. Assume (x_a, x_b, x_c x_d) primitives will be instrumented at location (a,b,c,d) respectively. Then, we could setup a system of `Linear Diophantine Equation` as follows.

$$\begin{cases} \Delta d_{dig} x_a + \Delta d_{fill} x_c + d_1 = 0 \\ \Delta d_{dig}(x_a + x_b) + \Delta d_{fill}(x_c + x_d) + d_2 = 0 \end{cases}$$

5.6.2 Equation for Multi-Object Layout Constraint

In general, for multi-object position constraint heap layout, we will generate a system of Diophantine equations as below:

$$\begin{cases} \Delta d_1 x_1 + \ldots \Delta d_m x_m + \Delta d_{a1} x_{a1} + \ldots \Delta d_{an} x_{an} + d_a = 0 \\ \Delta d_1 x_1 + \ldots \Delta d_m x_m + \Delta d_{b1} x_{b1} + \ldots \Delta d_{bn} x_{bn} + d_b = 0 \\ \ldots \end{cases}$$

In this system, each equation represents the constraint for one object, where, d_k (k=a, b, ...) are the `Target Distance` of each object constraint. x_k (k=1, 2, ..., m) are the count of instrumented primitives that can change the `Target Distance` of multiple objects, thus are shared between multiple Diophantine equations. x_{ak}, x_{bk} ... (k=1, 2, ..., n) are the count of instrumented primitives that only change `Target Distance` of one object, thus are not shared between equations.

5.6.3 Equation Decoupling

In some cases, different objects are indeed independent, and their equations could be decoupled from the system. As shown in Figure 5(e), after placing the first object `Malloc 3` at `Hole 1`, we could freely manipulate the object `Malloc 4` and place it to the `Hole 2`. In other words, they are independent and their equations can be decoupled.

Decoupling the Diophantine equations will greatly simplify the equations and reduce unknown side effects. We also proposed several techniques to adjust the order of target allocations and deallocations in program path, to decouple equations in the system as many as possible. Details could be found in the Appendix D.

5.7 Success Factors of Heap Manipulation

Heap layouts can not always be manipulated to the desired state. Few studies have analyzed the factors that affect the success rate of heap layout manipulation. Such studies can guide heap layout manipulation, not only for automated solutions but also for security experts.

5.7.1 One-Object Layout Manipulation

As shown in Theorem 1, a one-object layout constraint is solvable if there are at least one dig and one fill primitives, and all primitives' `Delta Distance`' greatest common divisor (gcd) divides the `Target Distance` of the object. Therefore, if there are no dig or no fill primitives, the success rate of heap manipulation is low.

Further, the gcd of all primitives' `Delta Distance` is also a key factor, since it should divide the `Target Distance`. Ideally, if the gcd is 1, then this equation is always solvable (assuming both dig and fill primitives exist).

Note that, if we have more primitives, it is more likely that their gcd will be smaller and even reach to 1. Therefore, we could infer that, the diversity of primitives is a key factor of the success rate. To improve the diversity, MAZE tries to discover as many primitive as possible, and analyze their semantics in detail to figure out their heap operation size (since different sizes yield primitives with different `Delta Distance`).

Existing techniques, such as SHRIKE, argued that the noise (i.e., extra heap (de)allocation in primitives) is the factor affecting the success rate. If the noise is 0, i.e., the `Delta Distance` of a primitive is 1, then primitives' gcd will be 1 and the equation is solvable. This confirms the high success rate of SHIKE and Gollum when the noise is 0. However, we pointed out that, noise itself is not the key factor. Primitives with many noises could still have a high success rate, as long as their gcd is a proper value (e.g., 1).

5.7.2 Multi-Object Layout Manipulation

As explained in §5.6, a multi-object layout constraint equals to a system of Diophantine equations. Each equation itself should be solvable. Therefore, the diversity of primitives is also an important factor for multi-object layout manipulation.

Moreover, all equations should be solvable together. If a linear combination of these equations yields an equation with only dig (or fill) primitives, then this system of equations in general has no solutions. This case is denoted as `equation entanglement`, usually caused by the following two reasons.

Inconsistency between the hole creation order and the object allocation order. In the POC path, if two target objects are allocated in a specific order, but their target holes are created in an inconsistent order, then in general the layout constraint has no solutions.

As shown in Figure 5(c), the object `Malloc 4` is allocate before `Malloc 3`, but its target hole `hole 2` is created after `hole 1`, this layout cannot be satisfied assuming the underlying heap allocator adopts a FIFO policy. The system of Diophantine equations is as below:

$$\begin{cases} \Delta d_{dig} x_a + \Delta d_{fill} x_c + \Delta d_{fill} x_d + d_1 = 0 \\ \Delta d_{dig} x_a + \Delta d_{dig} x_b + \Delta d_{fill} x_c + d_2 = 0 \end{cases}$$

Any solution to the first equation will fix x_a, x_c and x_d, and transforms the second equation to the following form

$$\Delta d_{dig} x_b + d^{'} = 0$$

Since there is no fill primitive in this equation, so it usually has no positive integer solutions, unless Δd_{dig} and $d^{'}$ has different signedness and the latter is a multiple of the former.

SHRIKE [8] demonstrated that the order of allocation relative to the memory corruption direction influenced the success rate. It is a heuristic speculation, not the real reason.

Lack of instrumentation points. In some cases, two target holes are created in one primitive, as shown in Figure 5(d). Then, there is only one instrumentation point available for dig primitives, no matter what objects they are used for. As a result, the system of Diophantine equations looks like:

$$\begin{cases} \Delta d_{dig}x_a + \Delta d_{fill}x_c + d_1 = 0 \\ \Delta d_{dig}x_a + \Delta d_{fill}x_c + \Delta d_{fill}x_d + d_2 = 0 \end{cases}$$

Any solution to the first equation will fix x_a and x_c, and transforms the second equation to the following form

$$\Delta d_{fill}x_d + d^{'} = 0,$$

which does not have positive solutions in many cases.

Similarly, if two target objects are allocated in one primitive, there is only one instrumentation point available for fill primitives. It also causes trouble for heap layout manipulation.

6 EVALUATION

We implemented a prototype of MAZE based on the binary analysis engine S2E [13]. It has over 16K lines of code to extract heap layout primitives and analyze their semantics, and over 12K lines of code to infer the desirable heap layout interaction sequence. Then, we evaluated its performance in a Ubuntu 17.04 system running on a server with 115G RAM and Intel Xeon (R) CPU E5-2620 @ 2.40GHz*24.

6.1 Result Overview

We evaluated MAZE in the following three different settings. All programs are tested in a regular modern Linux operating system (Ubuntu 17.04), with the defense DEP [37] and ASLR [38] enabled.

CTF benchmarks: We evaluated MAZE against 23 vulnerable programs collected from 20 CTF competition, most of them can be found in CTFTIME [39].

Out of 23 programs, MAZE can hijack control flow for 5, leak arbitrary memory address information for 1, and successfully generate an exploitable heap layout for another 10. But it also failed to manipulate the heap layout for 7 programs.

PHP benchmark: We collected 5 public PHP vulnerabilities (CVE-2013-2110, CVE-2015-8865, CVE-2016-5093, CVE-2016-7126 and CVE-2018-10549) and used their overflowed buffers as source objects. And then, we selected 10 data structures with exploitable data fields (e.g., code pointers) and use them as destination objects. By pairing each source object with destination object, we could get 50 expected heap layouts, in which the destination object is placed right after the source object. This setting is same as Gollum [9].

Table 1: CTF programs successfully processed by MAZE.

Name	CTF	Vul Type	Final State
sword	PicoCTF '18	UAF	EIP hijack
hacknote	Pwnable.tw	UAF	EIP hijack
fheap	HCTF '16	UAF	EIP hijack
main	RHme3 CTF '17	UAF	Memory write
cat	ASIS Qual '18	Double free	Memory write
asvdb	ASIS Final '18	Double free	Memory leak
note3	ZCTF '16	Heap bof	Unlink attack
stkof	HITCON '14	Heap bof	Unlink attack
Secure-Key-Manager	SECCON '17	Heap bof	Unlink attack
RNote2	RCTF '17	Heap bof	Unlink attack
babyheap	RCTF '18	Off-by-one	Unlink attack
secret-of-my-heart	Pwnable.tw	Off-by-one	Unlink attack
Mem0	ASIS Final '18	Off-by-one	Unlink attack
quotes_list	FireShell '19	Off-by-one	Unlink attack
freenote	0CTF '15	Double free	Unlink attack
databank	Bsides Delhi	UAF	fastbin attack

Table 2: CTF programs that MAZE failed to exploit.

Name	CTF	Vul Type	Failed Reason
multi-heap	TokyoWesterns	UAF	Multi thread
SimpleGC	34c3	UAF	Multi thread
vote	N1CTF '18	UAF	Multi thread
Auir	CSAW '17	UAF	Path explosion
Secret Note V2	HITCON '18	Heap bof	Path explosion
jmper	SECCON '16	Off-by-one	Path explosion
video-player	SECCON '17	UAF	Random layout

MAZE can generate all expected layouts in 68 seconds, far faster than SHRIKE and Gollum. What's more, MAZE is fully automated. By comparison, both SHRIKE and Gollum need a template provided by security experts to guide the heap layout manipulation process.

Python and Perl benchmark: We evaluated MAZE on Python and Perl. And MAZE can solve all the 10 vulnerabilities within 2 minutes.

Synthetic benchmarks: To thoroughly evaluate MAZE's `Dig & Fill` algorithm against other solutions, we referred to the synthetic benchmarks used in SHRIKE [8]. Besides layout noise, we added more factors to evaluate how they impact the effectiveness of layout manipulation. We evaluated MAZE against more than 3000 randomly generated test cases on two heap allocators: ptmalloc [25] and dlmalloc [26].

We evaluated the influence of noises. The result shows that if there are more than two primitives, the success rate remains at more than 95%, regardless of the number of noises.

We also evaluated MAZE against more complicated heap layout constraints which could even lead to nonlinear additivity of primitives. The result shows that it only impacts a proportion of different types of Diophantine equations. And the success rate remains at more than 90%.

In the end, we also evaluated MAZE against multi-object heap layout constraint for the heap allocator ptmalloc. The result shows that the success rate is still more than 95%.

6.2 CTF Benchmark

The details of all CTF programs evaluated by MAZE, are shown in Table 1 and 2.

Table 3: Heap layout primitives results on CTF programs.

Program	Paths	Symbolized Paths	Independent Primitives	Dependent Primitives	Time(s)
sword	118	11	5	5	500
hacknote	8	5	3	1	71
fheap	55	5	4	1	370
main	182	8	4	4	398
cat	44	10	4	5	1064
asvdb	7440	10	6	3	1156
note3	198	6	4	2	942
stkof	30	11	1	3	267
babyheap	18	6	3	2	163
secret...	12	4	2	2	186
Mem0	183	11	8	3	1099
Secure...	1332	55	5	3	445
quotes...	98	5	2	3	149
freenote	1068	7	3	4	1643
RNote2	62	6	3	3	359
databank	100	11	9	2	192

Table 4: Result of primitives assembly on CTF programs.

Program	Primitive Count	Noise Count	Constraint Count	D.a.F Time	POC Time	Solve Time
sword	4	0	1	5	26	1109
hacknote	32	1	1	58	14	406
fheap	17	1	1	26	38	3945
main	24	1	1	41	26	1046
cat	3	2	1	30	42	1013
asvdb	3	2	1	6	22	3105
note3	2	0	2	9	29	2600
stkof	4	0	3	19	33	1143
babyheap	9	0	2	14	24	2805
secret...	10	0	3	18	8	7646
Mem0	10	0	2	16	31	4974
Secure...	5	0	2	6	22	1676
quotes...	3	0	2	15	80	946
freenote	6	0	1	12	33	2034
RNote2	8	1	2	14	198	2779
databank	2	0	1	2	21	375

6.2.1 Successful Cases

Table 1 shows the list of programs successfully processed by MAZE. Out of 16 programs, MAZE can hijack control flow for 5 of them, and leak arbitrary memory address information for 1 of them. For the other 10 programs, MAZE only outputs the exploitable heap layout without generating exploits, since extra exploit techniques (e.g., `unlink attack`) are required to generate proper exploits but not supported yet.

6.2.2 Failed Cases

Table 2 shows the CTF programs that MAZE failed to generate expected layouts for. The major reasons are as follows:

- **Multi-Thread:** First, it's very difficult to analyze the dependence between different primitives in multi-thread applications. Second, race condition vulnerabilities between threads cause great difficulties to symbolic execution.
- **Path Explosion:** Although MAZE utilizes `Path Symbolization` to prune unnecessary paths and merge similar paths, complicated programs can still cause path explosion. For example, `Secret Note V2` embeds an AES algorithm, and `auir` is obfuscated by ollvm [40].
- **Random layout:** As discussed in Section 2.3.2, the heap allocator's behavior must be deterministic. Otherwise, MAZE may fail to infer the heap interaction sequence. For instance, there are random amount memory holes in `video-player` program's layout.

6.2.3 Effectiveness of Primitives Analysis

Table 3 shows the analysis results of heap layout operation primitives on CTF programs. We can see that, the number of original program paths is very large. But after applying our `path symbolization` technique, 15 of 16 programs' paths are reduced to about 10 symbolized paths, as shown in column 3. The average rate of path simplification is 98.4%.

Among these symbolized reentrant paths, MAZE further analyzes these primitives' dependency. Some primitives are independent from others, as shown in column 4. Column 5 shows the number of primitives that depend on others and can be analyzed by MAZE.

The last column shows the total time interval used for extracting and analyzing these primitives. MAZE could finish analyzing all 16 programs in several minutes. The average time cost is 9.4 minutes (562.7 seconds).

6.2.4 Efficiency of Primitives Assembly

Table 4 shows the evaluation result of the heap layout primitives assembly process. Column 2 shows the total number of available primitives, different from the number of symbolized paths listed in Table 3. MAZE will analyze the semantics of each symbolized path, and remove paths that cannot be used as primitives. It also analyzes the size of heap operations, and may yield multiple primitives for one symbolized path (with different allocation size). As shown in the table, MAZE could find at least 2 primitives for all 16 programs.

Some primitives may have more than one allocations and deallocations. The extra noise (de)allocations could cause trouble for heap layout manipulation, as argued in previous work, e.g., SHRIKE and Gollum. Column 3 shows the average number of noises in these primitives.

Column 4 shows the number of heap layout constraints to satisfy. There are 8 programs with one constraint. All of them have UAF or double free vulnerabilities, requiring to place one object at one location. Another 8 programs with two constraints all have buffer overflow vulnerabilities, requiring to place the vulnerable objects as well as the victim objects at proper locations. The other 2 programs requires three object constraints to facilitate unlink attacks.

The last three columns are the time cost, including the time used by the `Dig & Fill` algorithm (distance evaluation and equation solving), by POC analysis (vulnerability analysis and instrumentation points analysis), and by constraint solving (satisfying the final edited path). All steps are relatively fast, except the last constraint solving step, due to challenges to symbolic execution (e.g., loops and symbolic addresses).

6.3 PHP Benchmarks

To compare with existing solutions Shrike[8] and Gollum[9], we chose PHP as a real world target to evalu-

Table 5: Evaluation results of different solutions on PHP.

Solution	Solve time(s)	Succ	POC analysis time(s)
Maze	100% in 68s	100%	922s
Shrike	25% in 300s, 60% in 3000+s	60%	Not Supported
Gollum	75% in 300s, 85% in 2500+s	85%	Not Supported

ate. To trigger all the 5 vulnerabilities, we selected version 7.0.4. The evaluation result is shown in Table 5.

As shown in the second column, MAZE is much faster than Shike and Gollum. MAZE has solved all the benchmarks in 68 seconds. The average time consumption is only 27 seconds. Shrikes spent 300 seconds to solve 25% of them, and spent more than 3000 seconds to solve 60% of the benchmarks. Gollum solved 75% in 300 seconds and took more than 2500 seconds to solve 85%.

Further, MAZE can solve all the benchmarks. As a comparison, Shrike can only solve 60% of them, and Gollum solved 85%. After a more in-depth analysis, we figured out SHRIKE and Gollum failed mostly because of noises in heap primitives. Specifically, for CVE-2016-7126, the source buffer for this vulnerability is of size 0x20. There are many objects of size 0x20 in PHP, causing many noises in the POC path and the heap primitives and lowering the success rate of SHRIKE and Gollum. MAZE utilizes `Linear Diophantine Equation` to bypass the noise problem, regardless of the fact that all primitives have at least one noise.

Thirdly, both Shrike and Gollum need a template provided by security experts, to guide where to insert memory allocations and deallocations, as well as the allocation size. But MAZE is fully automated. It can analyze the POC and determine the layout state, as well as whether fill or dig operations are needed and where are the suitable instrumentation points.

Table 6: Evaluation results on Python and Perl.

Target	Vulnerabilities	Average time(s)
Python	CVE-2007-4965, 2014-1912, Issue24105, 24095, 24094	100% in 118s
Perl	Issue132544, 130703, 130321, 129024, 129012	100% in 141s

6.4 Python and Perl Benchmarks

To further evaluate the effectiveness, we also evaluated MAZE on Python and Perl. We chose 10 vulnerabilities in Python and Perl, and showed the evaluation result in Table 6.

Compared with Gollum [9], MAZE supports both Python and Perl. It demonstrates that MAZE broadly extends the application scope of Gollum. What's more, as shown in the third column, MAZE can generate expected heap layouts for all the vulnerabilities, and is much faster than Gollum.

6.5 Synthetic Benchmarks

We further utilize synthetic benchmarks to perform flexible and scalable evaluation of the `Dig & Fill` algorithm, to discover factors that can influence its performance.

To compare with other algorithms, we extended SHRIKE's benchmark with some modifications so that it can be adapted

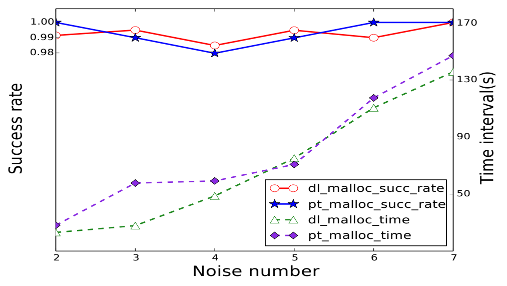


Figure 6: Influences of different number of noises.

to MAZE. First, MAZE generates heap layout primitives randomly, each primitive contains variable amount of allocation or deallocation operations (i.e., noises). Then, MAZE combines these primitives randomly to derive the initial heap layout. Third, MAZE randomly selects some memory holes and allocation operations, expecting a layout that the selected allocation operation takes the selected memory hole. Finally, MAZE utilizes `Dig & Fill` algorithm to calculate a heap interaction sequence to yield the expected layout.

6.5.1 Benchmark Setup

Besides the layout noise, we also tested other parameters that may affect the success rate of heap layout manipulation.

- **Noise number:** It's the minimum amount of noise operations placed in each primitive. Primitives could have more noises than this threshold.
- **(De)allocation primitives count:** It's the number of randomly generated primitives for heap (de)allocation. This factor represents the diversity of primitives.
- **Size list:** It represents the diversity of the size of allocation operations in a primitive. Allocation operations in each primitive will select one size from this list. The probability of selecting each size is also adjustable.
- **Mix of allocation and deallocation:** It indicates that the relative rate of heap allocation and deallocations in one primitive. If this factor is None, each primitive can only contain allocations or deallocations, but not both.

6.5.2 Evaluation of One-Object Layout Constraint

As aforementioned, a multi-object layout constraint can be transformed into multiple one-object layout constraints. So we will fully evaluate one-object layout constraint at first.

Factors Influencing Success Rate. SHRIKE demonstrated that the noise impacts the success rate. For instance, a single noisy allocation can make the success rate drop to 50% across all allocators. But as discussed in Section 5, the diversity of heap layout primitives is the major factor that influences the success rate of `Dig & Fill`, not the number of noise. We will prove this with a concrete experiment.

Influence of noise count. First, we evaluated the success rate of `Dig & Fill` using primitives with different noise count. In this evaluation, the noise number ranges from 2

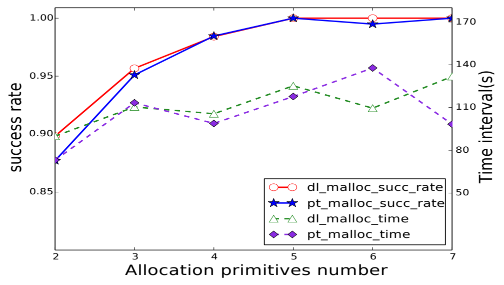


Figure 7: Influences of different number of primitives.

to 7. To exclude the influence of other factors, the number of (de)allocation and deallocation primitives is fixed to 3(4), the length of the heap operation size list is 1, and the mix of allocation and deallocation is None. For each setting, we generated 200 random test case.

The result is shown in Figure 6, solid lines are the success rates for different number of noises, while dotted lines represent the time cost. We can see that, the success rate keeps between 98% and 100%, showing that the number of noises does not influence the success rate of `Dig & Fill`. Furthermore, the time cost increases along with the number of noises, since noises will make the heap layout more complicated and cost more time to solve them.

Influence of primitive count. Then, we evaluated the success rate using a different number of primitives. In this evaluation, the allocation primitives count ranges from 2 to 7, and the number of deallocation primitives is set to 1, and the noise number is 5. Other configurations are the same as above.

The result is shown in Figure 7, the solid lines are the success rate for different number of allocation primitives, while the dotted lines represents the time interval spent to solve the problem. We can see that, as the number of primitives increases, the success rate also increases. This proves that the diversity of primitives influences the success rate. But even with only two primitives, the success rate can still reach 87.7%. Further, the time spent by MAZE to solve the problem does not grow along with the number of primitives.

Table 7: The success rate and time interval in different nonlinear additivity situations.

Target	Mix	Size Diversity	Mix + Diversity
pt_malloc	94.7% in 256s	98.9% in 384s	99.1% in 357s
dl_malloc	97.8% in 327s	100% in 433s	100% in 446s

Influence of Nonlinear Additivity. As discussed in Section 5.4.2, to handle the nonlinear additive factors, MAZE utilizes grouping, correcting and shifting techniques. For complicated heap layouts, MAZE can only generate two-variable or half Diophantine equations.

Table 7 shows the success rate and average time interval spent for primitives without linear additivity. Although MAZE can only generate half Diophantine equations, but the success rate of ptmalloc and dlmalloc are both more than 94% in all the nonlinear additivity situations. The biggest impact of nonlinear additivity is the time cost. Because MAZE can not derive the heap interaction sequence by solving equations, so it will spend more time for half Diophantine equations. Even so, the average time interval is still lower than 10 minutes. More detail of this experiment can be found in Appendix E.

Table 8: Results of multi-object layout constraint evaluation.

Target	Object count	Time (s)	Success rate	Nature	Reversed
PT	2	73.1	98.0%	72.1%	27.9%
PT	3	95.2	97.0%	55.1%	44.9%
PT	4	145.6	96.4%	52.2%	47.8%
PT	5	238.8	95.6%	50.4%	49.6%

6.5.3 Evaluation of Multi-object Position Constraint

Table 8 shows the evaluation result of multi-object layout constraint solving. We set the noise to 3 and (de)allocation primitive number to 3(4) respectively, and generate 100 random heap layouts for each multi-object constraint. We evaluated 2 to 5 object constraints, and the result shows that the success rate is more than 95% for all of them.

The success rate decreased and the time interval increased, while the number of objects increases. The root cause is simple. With more object layout constraints, MAZE has to generate more `Diophantine Equations` to solve.

SHRIKE demonstrated that the order of allocation relative to memory corruption direction also influenced the success rate. We also evaluated this factor. In the last two columns, the column `Nature` shows the ratio of cases, in which an earlier allocation takes the lower memory address but a later allocation takes the higher address, and the column `Reversed` shows the contrary. Because the heap layout is randomly generated, the Nature ratio drops when more objects layout constraints are enforced. For 5 object constraints, the Nature ratio is even 50%, but the success rate can still be 95.6%. So this factor has few influences on the success rate.

7 Discussion of Scalability

Dig & Fill algorithm First of all, regardless of what the applications are, `Dig & Fill` algorithm's scalability is only related to the adopted heap allocators. We have evaluated the scalability of `Dig & Fill` in Section 6.5 with test cases which are much more complicated than real world situation. And the result shows that MAZE can solve more than 90% of scenarios in minutes.

Some heap allocators (e.g. allocators in V8) utilize lots of security mechanisms to increase the difficulty of memory layout manipulation. For example, the OldSpace and the NewSpace mechanism makes it impossible to dig memory holes and place noise objects, causing troubles even for human analysts. Moreover, these allocators also violate the four rules defined in Section 2.3.2. So they are out of the scope of MAZE. We will try to address these advanced security mechanisms in the future.

Heap layout primitive analysis Like many other solutions, MAZE can also handle interpreters, such as PHP, Python and Perl. It's very challenging to handle programs whose inputs can not be freely assembled, such as network programs. MAZE utilizes symbolic execution to extract and analyze heap primitives for such programs. But due to the well-known bottleneck of symbolic execution, the current prototype of MAZE is not evaluated on complicated network services. Instead, we evaluated MAZE on CTF applications, which have similar process logic, complicated allocators, and compact input format requirements as network services.

Even for complicated network services, if its heap layout primitives are provided to MAZE (e.g., by human), MAZE can still generate the expected memory layout using its `Dig & Fill` algorithm.

8 Conclusion

Few AEG solutions are able to manipulate heap layouts into an expected state. We proposed a solution MAZE to transform POC samples' heap layouts into expected layouts and automatically generate working exploits when possible. MAZE extends heap layout primitives to reentrant code snippets in event loop driven applications, and could efficiently recognize and analyze them. MAZE further adopts a novel `Dig & Fill` algorithm to assemble primitives to generate expected layout, by deterministically solving a `Linear Diophantine Equation`. It is very efficient and effective, comparing to existing solutions, and even supports multi-object constraints and many heap allocators. Beyond heap layout manipulation, AEG has a lot of other challenges to address.

Acknowledgement

This work was supported in part by Beijing Municipal Science and Technology Project (No.Z181100002718002), National Natural Science Foundation of China under Grant 61772308, 61972224 and U1736209, and BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009.

References

[1] S. Heelan, "Automatic generation of control flow hijacking exploits for software vulnerabilities," Ph.D. dissertation, University of Oxford, 2009.

[2] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *Network and Distributed System Security Symposium*, 2011.

[3] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 380–394.

[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in *USENIX Security Symposium*, 2011, pp. 25–41.

[5] S.-K. Huang, M.-H. Huang, P.-Y. Huang, C.-W. Lai, H.-L. Lu, and W.-M. Leong, "Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 78–87.

[6] "Cve details," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.cvedetails.com/

[7] A. Sotirov, "Heap feng shui in javascript," *Black Hat Europe*, 2007.

[8] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 763–779.

[9] ——, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1689–1706.

[10] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1707–1722.

[11] "Unlink Exploit ," https://heap-exploitation.dhavalkapil.com/attacks/unlink_exploit.html, 2018, online: accessed 01-May-2018.

[12] "Diophantine equation," https://en.wikipedia.org/wiki/Diophantine_equation, 2019, online: accessed 01-May-2019.

[13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] D. Repel, J. Kinder, and L. Cavallaro, "Modular synthesis of heap exploits," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, 2017, pp. 25–35.

[15] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1914–1927.

[16] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 99–116.

[17] B. Garmany, M. Stoffel, R. Gawlik, P. Koppe, T. Blazytko, and T. Holz, "Towards automated generation of exploitation primitives for web browsers," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 300–312.

[18] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "Fuze: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[19] W. Wu, Y. Chen, X. Xing, and W. Zou, "{KEPLER}: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1187–1204.

[20] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to auto-

matically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[21] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits." in *USENIX Security Symposium*, 2015, pp. 177–192.

[22] K. Ispoglou, B. Albassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," 2018.

[23] J. Vanegue, "The automated exploitation grand challenge," in *presented at H2HC Conference*, 2013.

[24] ——, "The automated exploitation grand challenge, a five-year retrospective," in *IEEE Security & Privacy Langsec Workshop*, 2018.

[25] "The gnu c library (glibc)," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://www.gnu.org/software/libc/

[26] "A memory allocator by doug lea," 2019, online: accessed 26-Feb-2019. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html

[27] P. Argyroudis and C. Karamitas, "Exploiting the jemalloc memory allocator: Owning firefox?s heap," *Blackhat USA*, 2012.

[28] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with JavaScript," in *Workshop on Offensive Technologies (WOOT)*, 2008.

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *the 2012 USENIX Annual Technical Conference*, 2012, pp. 309–318.

[30] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 2015, pp. 46–55.

[31] A. Samsonov and K. Serebryany, "New features in addresssanitizer," 2013.

[32] "Dataflowsanitizer," https://clang.llvm.org/docs/DataFlowSanitizerDesign.html, 2018, online: accessed 01-May-2018.

[33] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[34] T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *International Static Analysis Symposium*. Springer, 2007, pp. 170–183.

[35] L. Cojocar, T. Kroes, and H. Bos, "Jtr: A binary solution for switch-case recovery," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2017, pp. 177–195.

[36] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," *Science of Computer Programming*, vol. 40, no. 2-3, pp. 171–188, 2001.

[37] S. Andersen and V. Abella, "Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies," http://technet.microsoft.com/en-us/library/bb457155.aspx, 2004.

[38] PaX-Team, "PaX ASLR (Address Space Layout Randomization)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[39] "Ctf time," https://ctftime.org, 2018, online: accessed 01-May-2018.

[40] "Obfuscator-llvm," 2019, online: accessed 26-Feb-2019. [Online]. Available: https://github.com/obfuscator-llvm/obfuscator

A Proof of Theorem 1

Bezout's Lemma. *A* `Linear Diophantine Equation`*:*

$$a_1x_1 + a_2x_2 + a_3x_3 + \ldots + a_nx_n = d$$

has an `integer solution` *$(x_1, x_2, \ldots, x_n)$, if and only if d is a multiple of the greatest common divisor* $\gcd(a_1, a_2, \ldots, a_n)$*.*

If there are at least one dig and one fill primitive, the `Linear Diophantine Equation` has positive and negative integers, and it's as below:

$$\Delta d_{f1}x_1 + \ldots\Delta d_{fn}x_n - \Delta d_{d1}y_1 - \ldots\Delta d_{dm}y_m + d = 0 \tag{1}$$

where $\Delta d_{f1}\ldots\Delta d_{fn} > 0$ and $+\Delta d_{fi}$(i = 1,2..n) is the `Delta Distance` of each fill primitive, $\Delta d_{d1}\ldots\Delta d_{dm} > 0$ and -Δd_{di} is the `Delta Distance` of each dig primitive.

According to the lemma, if gcd(d_{f1}, d_{f2}, ..., d_{fn},d_{d1}, d_{d2}, ..., d_{dm},) divides d, the `Linear Diophantine Equation` 1 have an integer solution, let the solution be $x_1^* \ldots x_n^*, y_1^* \ldots y_m^*$.

If there are integers x_{gi} and y_{gi}(i = 1,2..n) and

$$\Delta d_{f1}x_{g1} + \ldots\Delta d_{fn}x_{gn} - \Delta d_{d1}y_{g1} - \ldots\Delta d_{dm}y_{gm} = 0 \tag{2}$$

Equation 2 can be changed into:

$$\Delta d_{f1}x_{g1} + \ldots\Delta d_{fn}x_{gn} = \Delta d_{d1}y_{g1} + \ldots\Delta d_{dm}y_{gm} \tag{3}$$

For equation 3, if n ≤ m, we select $\forall x_{gi}$, and let x_{gi}=$\sum_{j=1}^{m-n+1} x_{gij}$, and equation 3 can be changed into:

$$\Delta d_{f1}x_{g1} + \ldots\Delta d_{fi}\sum_{j=1}^{m-n+1} x_{gij} + \ldots\Delta d_{fn}x_{gn} = \Delta d_{d1}y_{g1} + \ldots\Delta d_{dm}y_{gm} \tag{4}$$

For equation 4, the left and right side have the same number of terms. So we can select $\forall x_{gi}$ and $\forall y_{gi}$ from each side, and let x_{gi}=lcm(Δd_{fi}, Δd_{di})÷Δd_{fi}, and y_{gi}=lcm(Δd_{fi}, Δd_{di})÷Δd_{di},

Because $\Delta d_{f1}\ldots\Delta d_{fn} > 0$ and $\Delta d_{d1}\ldots\Delta d_{dm} > 0$, therefore x_{gi} > 0(i = 1,2...n) and y_{gi} > 0(i = 1,2...m).

Then `Linear Diophantine Equation` 1 has a general solution (where k are integers):

$$\begin{cases} x_1 = x_1^* + kx_{g1} \\ \ldots \\ x_n = x_n^* + kx_{gn} \\ y_1 = y_1^* + ky_{g1} \\ \ldots \\ y_m = y_m^* + ky_{gm} \end{cases}$$

Because $x_{gi} > 0$(i = 1,2...n) and $y_{gi} > 0$(i = 1,2...m), therefore no matter x_i^* and y_i^* are positive or negative, we can get a positive solution by increasing k.

B Linear Additivity of Dig and Fill Primitives

In this section, we assume that the size of O and P are equal. If their sizes are not equal, the cause of non-linear additivity may be different, but MAZE can still use the same techniques to derive new linear additive primitives.

If the Delta Distance of a dig and a fill primitive are d_1 and d_2, after MAZE inserted them into the POC path, the Target Distance changed from d to d + d_1 + d_2, then we call the two primitives are linearly additive. Only if all the primitives in Linear Diophantine Equation are linearly additive, the solution can be used to guide the combination of primitives and counteract the noise. But due to the complexity of allocators, especially the splitting and merging mechanism, not all the primitives are linearly additive. The non-linear additivity can be divided into two types:

Non-linearly accumulated with the same type of primitives: This type of non-linear additivity is caused by the mix of allocations and deallocations in one primitive. For example, if a deallocation sub-primitive of a dig primitive has allocation operations, they may wrongly fill other holes created by other dig primitives, so this type of dig primitives are not linearly accumulated with themselves.

Non-linearly accumulated with different types of primitives Many allocators support the splitting mechanism. If there is a memory hole that is larger than the size of an allocation, allocators usually split the bigger chunk into two smaller parts and return one part to service the allocation.

After an in-depth analysis, we found three types of heap operation mainly cause this type of non-linear additivity. 1) Bad allocation: its size is not equal to the target allocation O's. 2) Bad hole: its size is not equal to the target hole P's. 3) Little allocation: its size is less than half of P's.

It because, according to rule 3 in Section 2.3.2, if the memory hole's size is equal to the allocation request, it has a higher re-use priority. So bad holes in dig primitives will always have a lower priority than P. Therefore they can not place the target allocation O. It means that they have no contribution to Delta Distance. But according to rule 2 in Section 2.3.2, the bad allocations in fill primitives can be placed in any freed area, including the bad holes and the target hole P, so they can contribute to Delta Distance. If the bad allocations fill the bad holes, they do not have linear additivity.

For little allocation, due to the splitting mechanism in allocators, P will be cut into a smaller hole, and O can not be placed at P again, which means the hole is filled. So the Delta Distance is measured as +1. But if the primitive is added again, the little allocation will be placed at the rest part of P, i.e. Delta Distance is 0. Therefore this primitive does not have linear additivity.

To solve the above problems, MAZE utilizes three techniques:

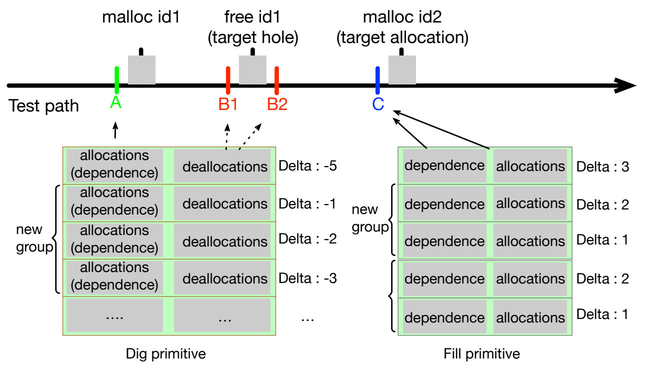


Figure 8: a Grouping technique example.

Grouping Grouping's core idea is to put multiple dig or fill primitives together and create a new primitive which is linearly accumulative with itself.

As shown in Figure 8, there is a dig primitive which is not linearly accumulated with itself. Then MAZE keeps inserting this dig primitive and the number of memory holes and allocations grows periodically, so the Delta Distance of this dig primitive will also change periodically. MAZE puts all the primitives in one cycle together and derives a new dig primitive. In Figure 8, one cycle includes three dig primitives, after grouping, Delta Distance of the new dig primitive group is -6 (-1-2-3), and the new dig primitive is linearly accumulative with itself. The same operation can also be applied for fill primitives.

Correcting MAZE also needs to correct the Target Distance and Delta Distance. For example, if there is a deallocation in a fill primitive and the allocator adopts a LIFO policy, the memory hole created by this fill primitive has a higher priority than the target hole P. So O will always be placed into this noise hole. To reach the expected heap layout, this memory hole and P must coincide, which means the noise allocation should be placed in P and then be freed to create a same memory hole as P.So Target Distance needs to be corrected.

MAZE corrects the Delta Distance and calculates a d_{fix} and generates a new Linear Diophantine Equation as below:

$$\begin{cases} \Delta d_1 x_1 + \dots + \Delta d_i x_i + \dots + \Delta d_n x_n + d + d_{fix} = 0 \\ x_1, x_2, x_3 \dots x_n \geq 0 \\ x_i > 0 \end{cases}$$

Shifting There are also bad allocations and bad holes in POC's execution trace. So after inserting a linear additive primitive, the actual change of Target Distance may not equal to the Delta Distance.

So MAZE utilizes the shifting technique to counteract the non-linear additive factors in POC. If there are bad or little allocations, MAZE keeps inserting dig primitives until all of them are placed in the newly created memory holes. Similarly, if there are `bad holes`, MAZE keeps inserting fill primitives to fill all of them. Then MAZE evaluates the new `Target Distance` and generates another `Linear Diophantine Equation`.

C Generate Diophantine Equations Based on Primitives' Linear Additivity

After grouping, correcting and shifting techniques, MAZE can derive new heap layout primitives which have better linear additivity, and then generate different types of `Linear Diophantine Equations`

C.1 Multi-Variable Diophantine Equation

If fill primitives do not contain `bad allocations` or `little allocations`. These fill primitives are linearly accumulated with almost arbitrary dig primitives. If MAZE can also find dig primitives that are linearly accumulated with dig primitives, then MAZE can generate a `Multi-variable Diophantine Equation` to derive the expected memory layout. Similarly, MAZE also searches dig primitives which contain no `bad hole`, and the following process is the same.

C.2 Two-Variable Diophantine Equation

If a fill pritimive contains `bad or little allocations`, or a dig primitive contains `bad holes`, but they are linearly accumulated with each other, MAZE will generate a `Two-Variable Diophantine Equation` for them. It may be extremely complicated to find three or more primitives that containts `bad allocations or holes` but are linearly accumulated with each other. It because MAZE has to enumerate all the possible permutations. So MAZE only generates `Two-Variable Diophantine Equation` for this situation.

C.3 Half Diophantine Equation

If all the primitives contains `bad or little allocations` or `bad holes`, and MAZE can not find a pair of dig and fill primitives that are accumulated with each other, MAZE will utilize the grouping technique again. In this grouping process, MAZE will select a pair of primitives and insert lots of dig primitives first, then keep inserting fill primitives and derives another new fill primitive group. This new fill primitive that the dig primitive are linearly additive now.

In this situation, it is almost impossible to measure the `Target Distance`, so MAZE will generate a `Half Diophantine Equation` without `Target Distance`. If the `Delta Distance` of the dig and the new fill primitive are coprime, which means that $\gcd(\Delta d_1, \Delta d_2) = 1$, it can always divide `Target Distance` and the `Linear Diophantine Equation` will always have solutions.

MAZE infers the solution of `Half Diophantine equation` in a novel way. If the `Target Distance` is greater than 0, MAZE keeps inserting dig primitives, and if the `Target Distance` is less than 0, MAZE keeps inserting fill primitives, until `O` is placed at `P`.

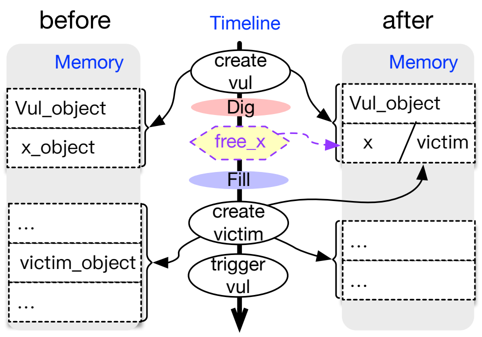


Figure 9: The example of locating a memory hole.

D Solutions to Equation Decoupling

For a specific heap layout, MAZE needs to find suitable positions, i.e. memory holes, and then utilizes `Dig & Fill` to place target objects at the appropriate target holes.

As discussed above, consistency between the hole creation order and the object allocation order affects the success rate, as well as the lack of primitive instrumentation points. Because of these factors, MAZE proposed two techniques to adjust the order of target allocations and target hole creations (deallocations), so that the system of Diophantine equations can be decoupled.

Locate Suitable Memory Holes Based on the vulnerability type and exploit technique, MAZE locates a potential target hole `P`. For example, to exploit a heap overflow vulnerability, a attacker needs to place a sensitive heap object next to the overflowed object. As shown in Figure 9, MAZE will locate the overflowed object (Vul_object) first. If the adjacent object (x_object) can be freed in another primitive (free_x) and the size is equal to the sensitive object (victim_object), MAZE will insert the deallocation primitive after the allocation of the overflowed object. Then MAZE utilizes `Dig & Fill` to generate a `Linear Diophantine Equation` for the sensitive object, because the fill and dig primitives' inserting point is after the allocation of the overflowed object, so that it will not affect its position. Similarly, MAZE can also locate the sensitive object first, then inserts a deallocation primitive to free the object ahead of it, to create a target hole for the overflowed object.

If the adjacent object can not be freed in another primitive, MAZE will try to find appropriate memory holes (e.g. contiguous holes) in the whole heap layout. If these memory holes can be freed in different heap primitives, MAZE can generate independent Diophantine equations by adjusting the `Dig & Fill` timing for each target object's allocation and target hole's deallocation. The core idea is to start a new `Dig & Fill` after the previous object's `Dig & Fill` is finished.

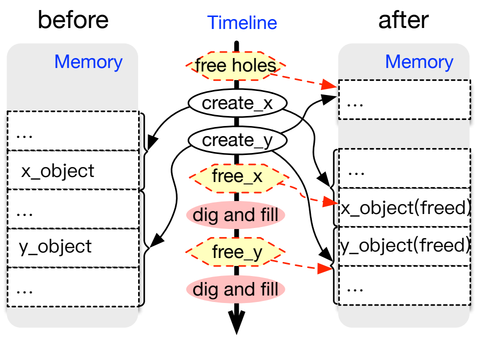


Figure 10: The example of creating memory holes.

Create Suitable Memory Holes If MAZE failed to locate suitable target holes, it would try to create them. First, MAZE searches for the heap primitive whose last allocation (size is equal to target object, i.e. overflowed object) can be freed in another primitive. For example, as shown in Figure 10, x_object's size is equal to the overflowed object, and it is the last allocation of create_x, it can also be freed in another primitive (free_x). Then MAZE searches for a primitive, which contains an allocation (size is equal to another target object, i.e. the sensitive object). In this example, it is the create_y. The next step is to insert create_x and create_y into the POC's beginning. Before the insertion, MAZE will utilize heap spray to fill all the holes in heap layout, so that all the objects allocated in create_x and create_y can be adjacent. Then MAZE will calculate the number of objects allocated before y_object in create_y and create the same amount of memory holes (the `free holes` hexagon in Figure 10) so that x_object and y_object can be adjacent. In the end, MAZE will free x_object, and then utilize `Dig & Fill` to generate a Diophantine equation for the overflowed object, after the object is placed at x_object's address, MAZE will use `Dig & Fill` to place the sensitive object. This solution can be applied to segregated storage allocators (tcmalloc, jemalloc) and boundary tag allocators (ptmalloc, dlmalloc).

If there is no such primitive, MAZE will create a memory hole with enough size to hold multi-objects. Then MAZE slightly changes the `Dig & Fill` algorithm to support placing an object to a memory hole with unequal size. The only difference is the cause of non-linear additivity. But MAZE can still use grouping, correcting and shifting techniques to derive new linear additive primitives. In the end, MAZE only needs to place objects to the big memory hole one by one by using `Dig & Fill`. This solution can be applied to boundary tag allocators that support the splitting mechanism, such as ptmalloc and dlmalloc.

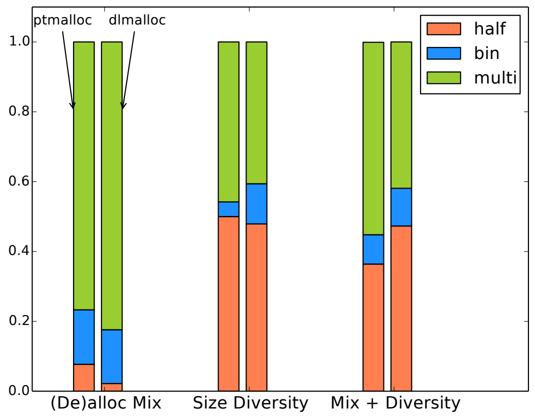


Figure 11: Proportion of multi-variable, two-variable and half `Linear Diophantine Equation` in different nonlinear additive situation.

E Influence of Nonlinear Additivity

The mix of allocations and deallocations and size diversity are non-linear additive factors. So we also evaluated the `Dig & Fill` algorithm on a more complex heap layout situation. In this evaluation, the allocation primitives count is 6, deallocation is 5, and the noise number is 5.

Mix of allocations and deallocations: We added 2 to 8 allocations to each dig primitive's deallocation part and 2 deallocations to each fill primitive and evaluated MAZE against 200 random test cases. The result is shown in Figure 11. Because all the allocations' size is the same, most of the generated Diophantine equations are multi-variable.

Size diversity: We added a size list that contains 5 random allocation sizes, and each primitive selected one size form this list randomly, the selection probability of each size is 5:2:1:1:1. The result is also shown in Figure 11. In this test, for almost 50% of the test cases, MAZE can only generate half Diophantine equations.

Mix + Size diversity: We put above two nonlinear additive factors together, and the result is also in Figure 11. MAZE still generated half Diophantine equations for 50% of the test cases. So we can conclude that the size diversity factor has more influence on the nonlinear additivity of primitives, and MAZE has to generate more half Diophantine equations.